BD-review

a platform for Music Reviews

Daniel Graziotin, 4801, daniel.graziotin@stud-inf.unibz.it

Summary

Application Domain

Music

Vision statement

The aim of the project is to build a dynamic website to allow people to review releases (albums, demos, EPs, singles) of (young, unsigned) music bands. Users will be able to signal interesting materials and review them, while other users will be able to comment the reviews, too.
This web 2.0-oriented application should allow unknown talented musicians to achieve a higher notoriety but also to improve their productions.

Technology overview

The System has been developed using

- J2EE technologies (JSP, Servlets and JavaBeans)
- Database support (PostgreSQL[1] 8.3) through JDBC 4[2]
- XHTML Strict 1.0 + Cascading Style Sheets 2.1 for presentation
- Apache Commons[3] for conversion and Bean population routines
- Some utility methods found on Books and Internet (their provenience is cited in the sourcecode)
- Javascript for confirmation system and form validation
- TinyMCE[4] rich WYSIWYG HTML editor

It has been tested with the most popular browsers (and rendering engines):

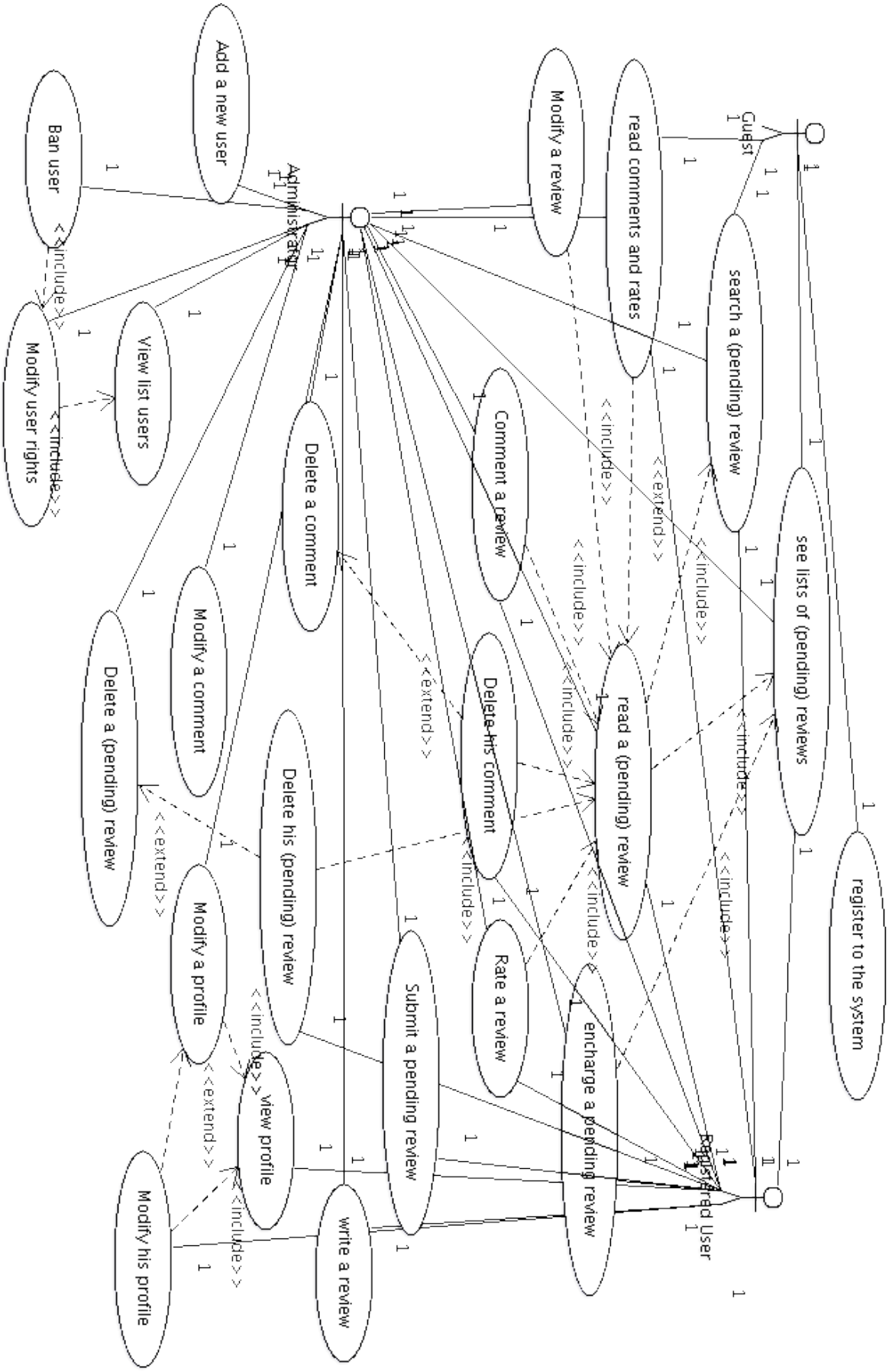
- Mozilla Firefox 3.0.10
- Internet Explorer 8
- Chromium 3.0.183.0

Software Engineering Outcomes

BD-review has been developed following the key phases of each software development process. Due to lack of time, a waterfall model has been used. The following pages contain the results of the analysis and design phases in form of tables and UML diagrams.

1 http://www.postgresql.org/
2 http://jdbc.postgresql.org/
3 http://commons.apache.org/components.html
4 http://tinymce.moxiecode.com/

Use Case Diagram



Picture 1: Use Case diagram for BD-review

Requirements

The following is the list of the functional requirements of the system. Besides 21,22,27 and 28, all the requirements have been implemented.

This can also be considered as a compact list of functions of the system.

ID	Requirement	Priority
		1 = must → 3 = nice
	Guest	
1	A guest can read a review	1
2	A guest can search through reviews	1
3	A guest can search artists	2
4	A guest cannot comment a review	1
5	A guest cannot rate a review	1
6	A guest can read comments and rates of a review	1
7	A guest sees a list the last published reviews	1
8	A guest can register himself to the system	1
	Registered User (= an extension of a Guest)	
9	A registered user sees a personalized page when he logins	1
10	A registered user knows the last time he visited the system	2
11	A registered user sees the list of reviews published from his last visit	2
12	A registered user can write a review from scratch	1
13	A registered user can not modify his review	1
14	A registered user can delete only his review	1
15	A registered user cannot modify a review	1
16	A registered user can write a response to a review (i.e. comment a review)	1
17	A registered user can delete his comment	1
18	A registered user can not modify his comment	1
19	A registered user can rate a review	1
20	A registered user can modify his personal details and password.	1
21	A registered user can see personal details of another registered user.	3
22	A registered user can choose to publish his profile	3
	Administrator (= an extension of the registered user)	
23	An administrator can see a list of the registered users	1
24	An administrator can change rights of a registered user (define moderator	1
25	An administrator can modify personal details of a registered user	1
26	An administrator can delete a user	1
27	An administrator can ban a user from the system	3
28	An administrator can manually add a new registered user	3
29	An administrator can delete a comment	1
30	An administrator can modify a comment	1

At the beginning there was the idea to have pending Reviews, which is the possibility to flag an interesting music release that somebody could review in a second moment. This was dropped from the system because of lack of time.

Implementation Strategy

BD-review is my first Java web application. I come from a couple of years of web development using fresh web development frameworks and languages[5] like Python Django[6]. Among the comfortable abstraction that those web-frameworks provide, I appreciate their strong ORM approach.

Therefore, I decided to implement the System thinking in a "frameworkish" way, by separation of concerns and achieving a higher level of abstraction:

The system entities should be simple and independent from any other external factor, i.e. the database system or any validation rules.

In this way, the models of the system are defined isolated in a separate "logic unit" package. This enforces re-usability.

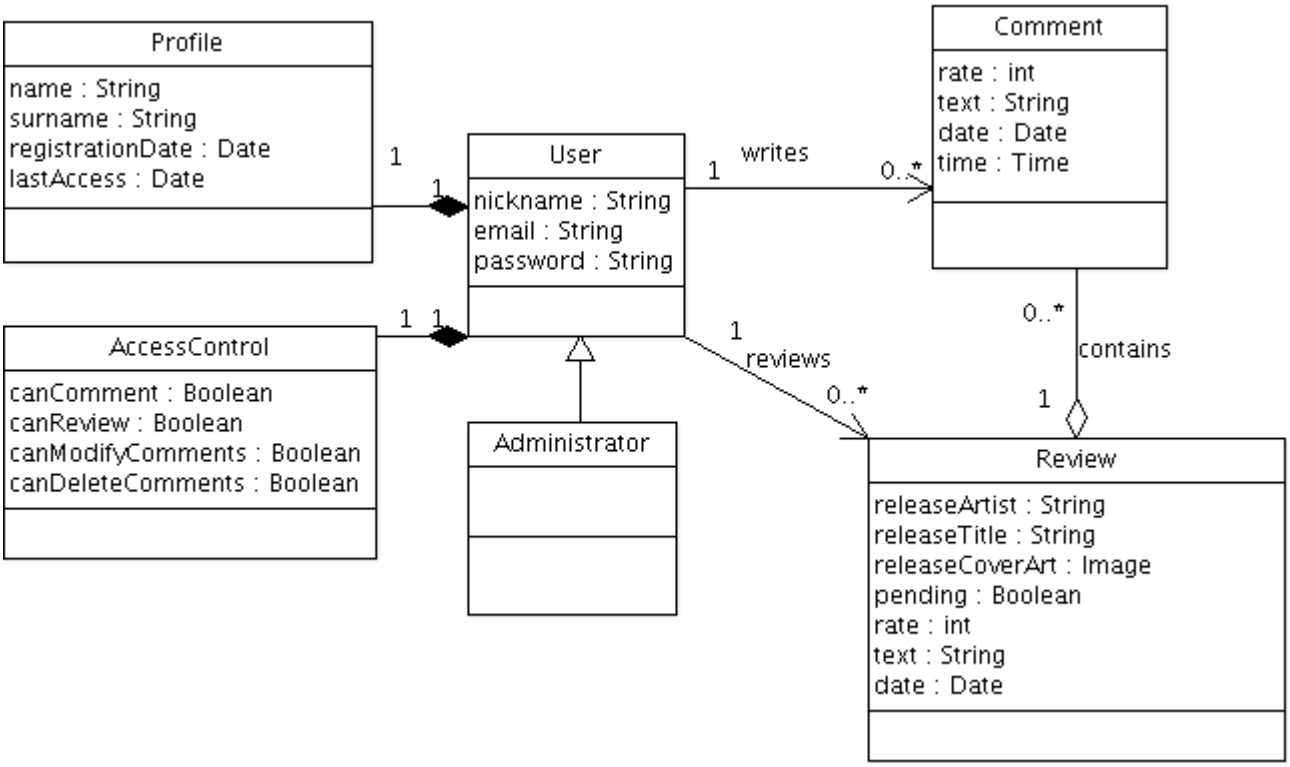
Another package should contain what I call the "database interfaces". These classes extends the models of the logic unit, adding a partial ORM support (I developed just the methods I needed) enclosing all the database related methods.
This permits to have things like Comment.getComments(User aUser) without writing queries in the Servlets. This also means that no queries should be done outside those database interfaces.

The architecture of the system is fully described on page 9.

5 Called web-frameworks from now on
6 http://www.djangoproject.com/

Design Class Diagram

My experience with web development using ORM-providing frameworks brought me to this first Design Class Diagram:



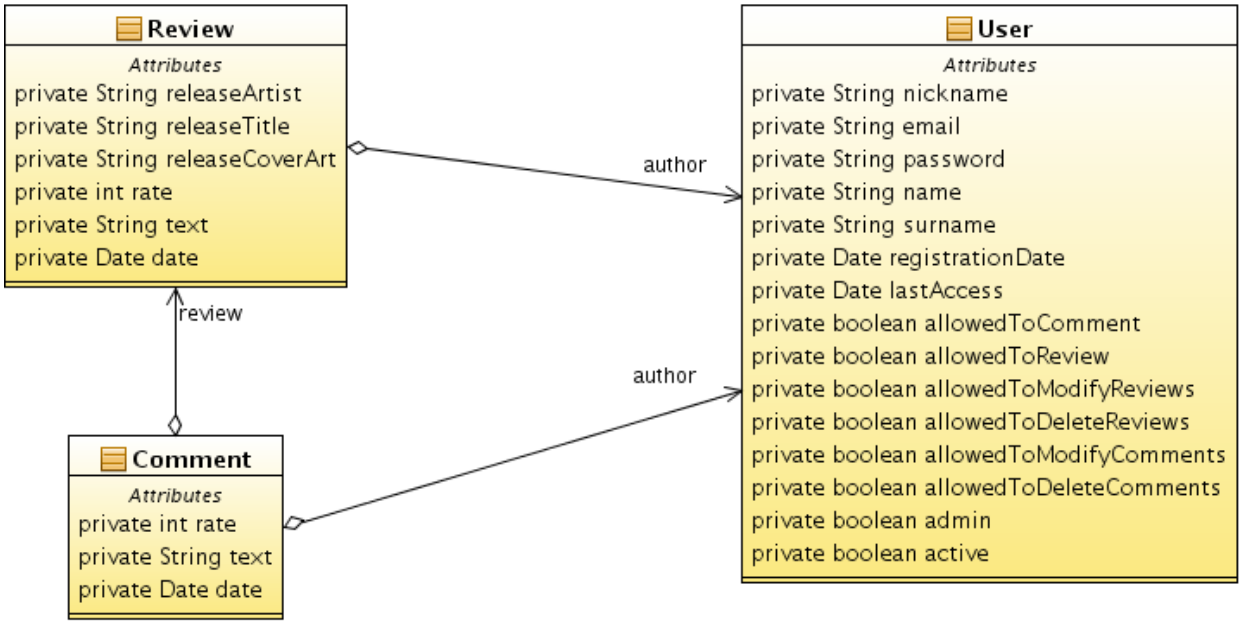
Picture 2: First Design Class Diagram

This can be considered a reasonable system design when using web development frameworks[7]. But after some days of development I realized that an elegant but more complex design would have been translated in to many classes and methods in the "database interfaces" package.

Therefore, I decided to switch to a less elegant but more practical design. I first merged User and Administrator in a single class named User. I also merged Profile in User and a couple of days after I also integrated *AccessControl* in User, too.

7 When using Django, the retrieval of the Profile of a user is done using user.get_profile() . See http://docs.djangoproject.com/en/dev/topics/auth/#django.contrib.auth.models.User.get_profile

The following is the final design of the logic unit. I omitted the operations because they are all made by getters, setters and constructors:



Picture 3: Final logic unit Class diagram

The reasons behind this decision are all related to my implementation strategy. If this was a real-world business application, I would have used the original design[8].

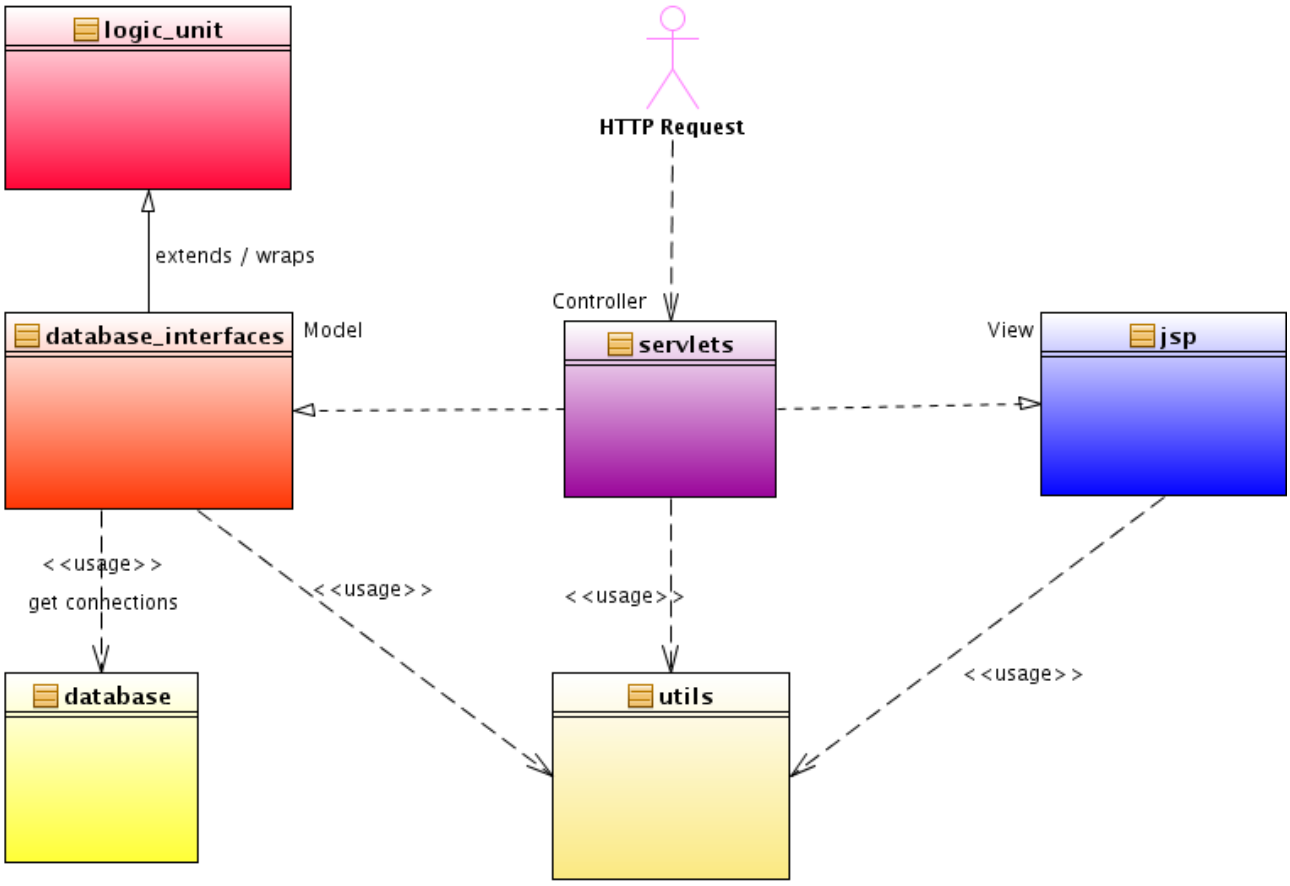
8 And maybe some frameworks like Java Server Faces or Spring

System Description

This section contains the description of how the system is structured. The first section contains the description of the packages and ends with the description of each Servlet. The other three sections are about user login and ACL, user input validation and the message system.

Architecture

BD-review is composed by 5 packages plus the JSP pages, that define layers and collaborations:



Picture 4: A model for the System Architecture and Component Interactions

it.unibz.bdreview.logicunit

Contains the JavaBeans that define the entities of the System.

- User.java
- Review.java
- Comment.java

it.unibz.bdreview.database

Classes and methods that define the database access and return connections.

- DBConnection.java – connects to the database and provides Connection objects statically[9].

it.unibz.bdreview.databaseinterfaces

Contains classes that extend those of the logic-unit package, acting as wrappers for database-related operations. All the methods to create, update, delete, obtain and link entities are defined in this package. This are to be considered the Models of the system.

- User.java
- Review.java
- Comment.java

it.unibz.bdreview.utils

Utility classes for user input validation, attribute string formatting for SQL queries, wrappers for o**rg.apache.commons.beanutils.BeanUtils.populate** etc.

- BeanUtils.java[10] - Acts as a wrapper to **org.apache.commons.beanutils.BeanUtils.populate(Object bean, Map propertyMap)**. Contains a *boolean emptyOrNull(String parameter)* utility method methods
- DBUtils.java – This class contains all static methods to help when dealing to the database. All the dbize and dbizeLast accept attributes of **it.unibz.bdreview.databaseinterfaces** classes and return them as a String suitable for queries. public static String quotify(String attribute) does also filter the parameter and escape the parameter, by calling the other two utility methods *public static String filter(String s)* and *public static String escape(String htmlString)*[11].

9 As written on the last page, this is not a good solution when developing multithread applications.

10 Taken from Core Servlets and JavaServer Pages 2nd Edition, http://volume1.coreservlets.com/archive/coreservlets/beans/BeanUtilities.java

11 User input validation is better described on page 15.

it.unibz.bdreview.servlets

Contains the servlets of the System, following MVC concepts. HTTP user requests are all made to the servlets of this package, which decide the actions to perform. The actions are made by using **it.unibz.bdreview.databaseinferfaces** and **it.unibz.bdreview.utils** classes and methods. The appropriate view is then called to present the output of the computation. The corresponding views are located in web and web/protected folders, with the same name of the Servlet (example: ReadReview.java has its view in web/readReview.jsp).

Servlets not in the /protected/ area:

- ReadReview.java
  Given an id parameter, constructs the corresponding Review object and stores it in the request, calling the view to visualize the results. An error message is displayed elsewhere.
- SearchReview.java
  Given the artist name and/or the release name (strings), it searches through the Reviews that are likely to be of user interest and stores them in a Vector. It calls the view to visualize the results. A "not found" message is displayed elsewhere. When not called with a POST request, it just calls the view to display the search form.
- RegisterUser.java
  Register a new user to the system, verifying that the nickname (=username) or the email are not already present. It creates a User object and stores it in the database when there are no error. It calls the view to inform the user about the success of the operation. When not called with a POST request, it just calls the view to display the registration form.
- LoginUser.java
  When called with a GET request, it presents the login form. Otherwise, it checks with the given nickname and password if there is a user that possesses the corresponding attributes. If successful, it stores the User object and his previous access timestamp in the Session. If fails, simply returns the login form via the view. If called with a "forwardURL" parameter, it also redirects to the corresponding URL. If requested by user, it also stores the username and the password in Cookies[12]
- LogoutUser.java
  Deletes the session attributes and Cookies related to the user and redirects to the home page.

12 Passwords are stored in plain text in cookies and in the database. This would not be a good solution and if this was a real life application. Passwords should be encrypted before stored.

Servlets in the /protected/ area:

Please note that all of the following servlets redirect to /LoginUser when there is no user logged in. They do also send 404 HTTP status when there are incorrect input parameters and 403 HTTP status when user is not authorized to perform the requested operations.

- UserPage.java
  Prepares the personal page of the logged user. It searches for the reviews stored after his last access. It searches the reviews written by the user. Presents the results via the view.
- ManageReview.java
  It searches for a "*do*" parameter in the request and lets user to write, edit or delete a review after a check to his rights. It calls the right view for each operation made and informs the user about the success or the fail of the operation requested.
- ManageComment.java
  It searches for a "*do*" parameter in the request and lets user to write, edit or delete a comment after a check to his rights. Besides for the comment edit, all the other operations sends a redirect to the requesting URL. The user is always informed about the result of his operation.
- ManageUser.java
  It works like ManageComment and ManageReview, using a "*do*" parameter in the request. Lets the user to edit his own profile[13], search for other users, delete a user, fully modify user details and access roles[14]. It calls the corresponding view to inform the user about the result of his operation, or to serve him the forms to ask for the operation.

Servlets directly accessible by a HTTP request:

- AccessControlFilter.java
  This filter provides access control for all requests in the application containing /protected/ in the URL, by looking for the authentication token in the session and forwarding to the login page if not found.
- RememberTheCookieFilter.java
  This filter intercepts all the HTTP requests and checks for the authorization cookie. If found, it registers the corresponding User object as a Session attribute.

Both the filters are better described in the next page.

13 It also checks if the new email set is already present in the database

14 It obviously checks the access right of the requesting user before doing any operation.

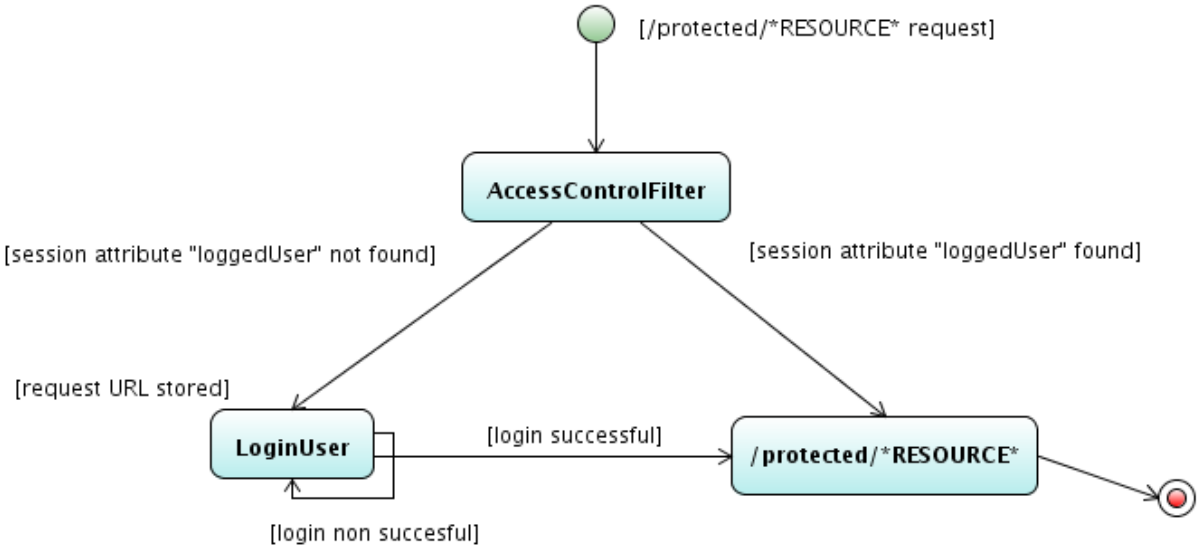
User Login and Access Control

User roles are determined by boolean flags. User roles are described in the requirements table.

A user logins using a classical login form. If the user login is successful, the corresponding User object is stored as an attribute of the Http Session. Also his last successful login date is stored as a session attribute.

- There are some system parts that only registered user can access.
- There are some system functions that only special users can access (those with diverse access rights).
- There is also a system part in which just administrator users can access.

All of this cases are special cases of a particular privileged access, the "logged user" access. So I thought that the easiest way to implement Access Control was to define a particular area of BD-review accessible only by logged users, the so called "protected area". I decided to follow the method described by Hans Bergsten[15]: a Servlet Filter.



Picture 5: Access to a protected resource

15 JavaServer Pages 3rd edition, O'Reilly, p. 385

The Servlet Filter **it.unibz.bdreview.servlets.AccessControlFilter**, a modified version of the one by Bergsten, intercepts requests to the protected area, searches for the "loggedUser" attribute in the session, forwards the request either to the appropriate protected Servlet if found or to the login page if not found.

Servlets under /protected/ are responsible to check User rights. This is a easy task since the presence of the User object in session is assured by the Filter.

it.unibz.bdreview.servlets.RememberTheCookieFilter Servlet is the last feature added to BD-review, to add Cookies support. The whole website and its protection system is built around requests and sessions attributes. Every page that needs to access to logged user information searches him in the session attribute. Therefore, the most intelligent way I could find was to add the feature as a layer, with another Servlet Filter that intercepts every HTTP request looking for the login Cookies set by LoginUser servlet. If found, it simply gets the corresponding User object and stores it as a session attribute, where the other servlets will try to find it.

Input Validation

To validate user input, three approaches were used:

1. The use of TinyMCE editor where possible
2. Filter every string submitted by the user, avoiding validation via Javascript. Filtering and escaping must be done server-side as Javascript can be manipulated (because it is executed by the client)

The second approach uses some utilities in **it.unibz.bdreview.utils**. For example, **DBUtils.filter(String s)** parses a string looking for HTML entities like *&*, *'*, *è*, *€* etc, and substitutes them with the corresponding &code;

Because of the possibility to use some HTML tags like *
* , *<p>*, ** via TinyMCE, I could not simply filter < and > characters for Review and Comment texts. Anyway, I had to find a way to stop <script> and <iframe> tags anyway, in case the user bypasses javascripts and deactivates TinyMCE.

So I wrote the **DBUtils.escape(String htmlString)** method, that parses the user user input using regular expressions. Every script tag is blocked and deleted, no matter how written:

```
public static String escape(String htmlString)
   {
      Pattern pattern = Pattern.compile("(<(/?)+script)",Pattern.CASE_INSENSITIVE);
      Matcher matcher = pattern.matcher(htmlString);
      String withoutScript = matcher.replaceAll("");
      pattern = Pattern.compile("(<(/?)+iframe)",Pattern.CASE_INSENSITIVE);
      matcher = pattern.matcher(withoutScript);
      String withoutIframe = matcher.replaceAll("");
      return org.apache.commons.lang.StringEscapeUtils.escapeSql(withoutIframe);
   }
```

This is a not very elegant solution but it works fine. At the end, the **org.apache.commons.lang.StringEscapeUtils.escapeSql(tmpString)** is also called to parse the string against possible SQL Injection attacks.

Example with input "*Lorèm Ipsùm <script>alert('looser!')</script>*":

```
DBUtils.dbizeLast("Lorèm Ipsùm <script>alert('looser!')</script>")
→ quotify("Lorèm Ipsùm <script>alert('looser!')</script>")
   → filter("Lorèm Ipsùm <script>alert('looser!')</script>")
      → escape("Lor&egrave;m Ips&ugrave; <script>alert('looser!')</script>")
return → "'" + "Lor&egrave;m Ips&ugrave; >alert('looser!')>" + "'"
```

DBUtils.vasteHTML(String htmlString) does simply substitute "<", ">", "'" and """ characters with the corresponding < and > codes. It is used where HTML is not necessary (ie. The name of the Artist).

User nickname and email is also validated server-side by using **DBUtils.hasValidCharacters(String s)** that checks if the given string matches a regular expression[16], depending if we are checking an email or a nickname. This method cannot be used with other fields such as artist name or release name because it is to strict.

16 I personally wrote the two regular expressions, therefore they are not perfect. But both work better than any method that checks each element of the string against illegal characters I could try out.

Message System

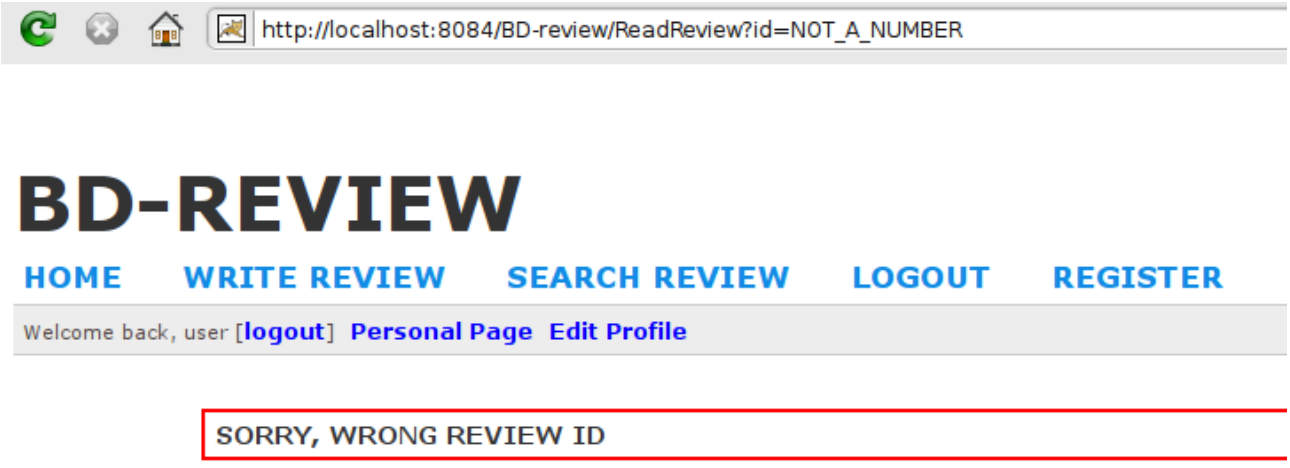
With so many user inputs and so many decision flows depending on parameter values, a good message system was needed. I decided to standardize simple messages and error messages thrown by the servlets by using attributes and parameters with name "*message*" in case of a simple notification and "*errorMessage*" in case of an error or a warning. It was easy then to define a method in *utils.jsp* that catches these messages and prints them to users:

```
<%!
   public String PrintMessages(HttpServletRequest request, HttpServletResponse response, String which) {
      StringBuffer messages = new StringBuffer("");
      if (which == null || which.equals("all")) {
         if (request.getAttribute("message") != null) {
            messages.append("<div class=\"message\">" + request.getAttribute("message").toString() + "</div>");
         }
         if (request.getAttribute("errorMessage") != null) {
            messages.append("<div class=\"errorMessage\">" + request.getAttribute("errorMessage").toString() +
"</div>");
[...]
         }
      } else {
[...]
         if (request.getParameter(which) != null) {
            messages.append("<div class=\"" + which + "\">" + request.getParameter(which).toString() + "</div>");
         }
      }
      return messages.toString();
   }
%>
```

This was also to experiment with JSP declarations.

In every page that has to print out some messages, it is as simple as a

```
<%= PrintMessages(request, response, "message")%>
```



Picture 6: A wrong Id input

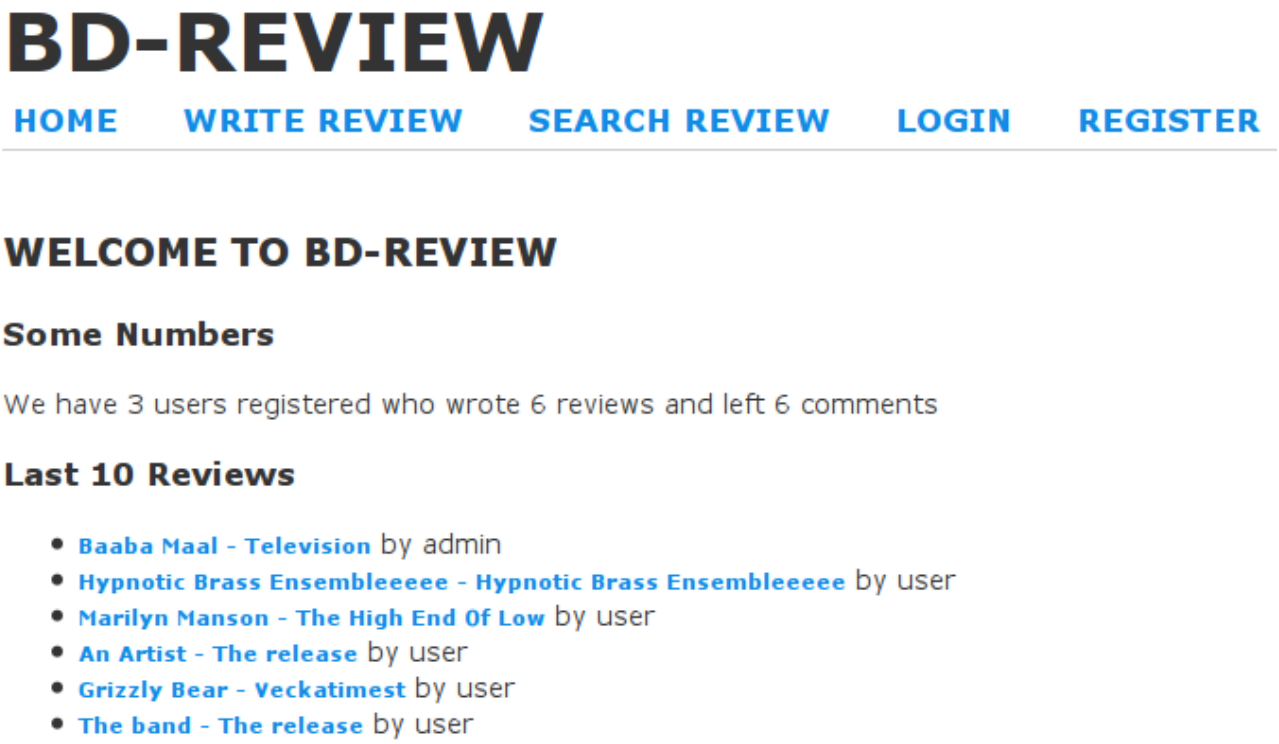
User Interface

In this section I will present some aspects of the user interface.

The layout

The presentation part is written using XHTML 1.0 Strict [17] and CSS 2.1.

I decided to have a very polished and minimal layout, composed by a single, headed column. The colors chosen are also limited. This is for focusing the user on the contents of the website:



Picture 7: A very minimal design

The website sections are clearly presented: on the top we can see the heading section, with the name of the site and the horizontal menu with just the most important sections. Everything else is a content of the website.

17 XHTML is tentatively valid. User input via TinyMCE is not completely controllable, as it lets user to copy and paste from other html files. This could obviously invalidate the code.

The User Page

Users know when they are logged thanks to some visible changes in the UI. In the following picture we can see the personal page of a user when he logins

Picture 8: Personal Page of a registered user

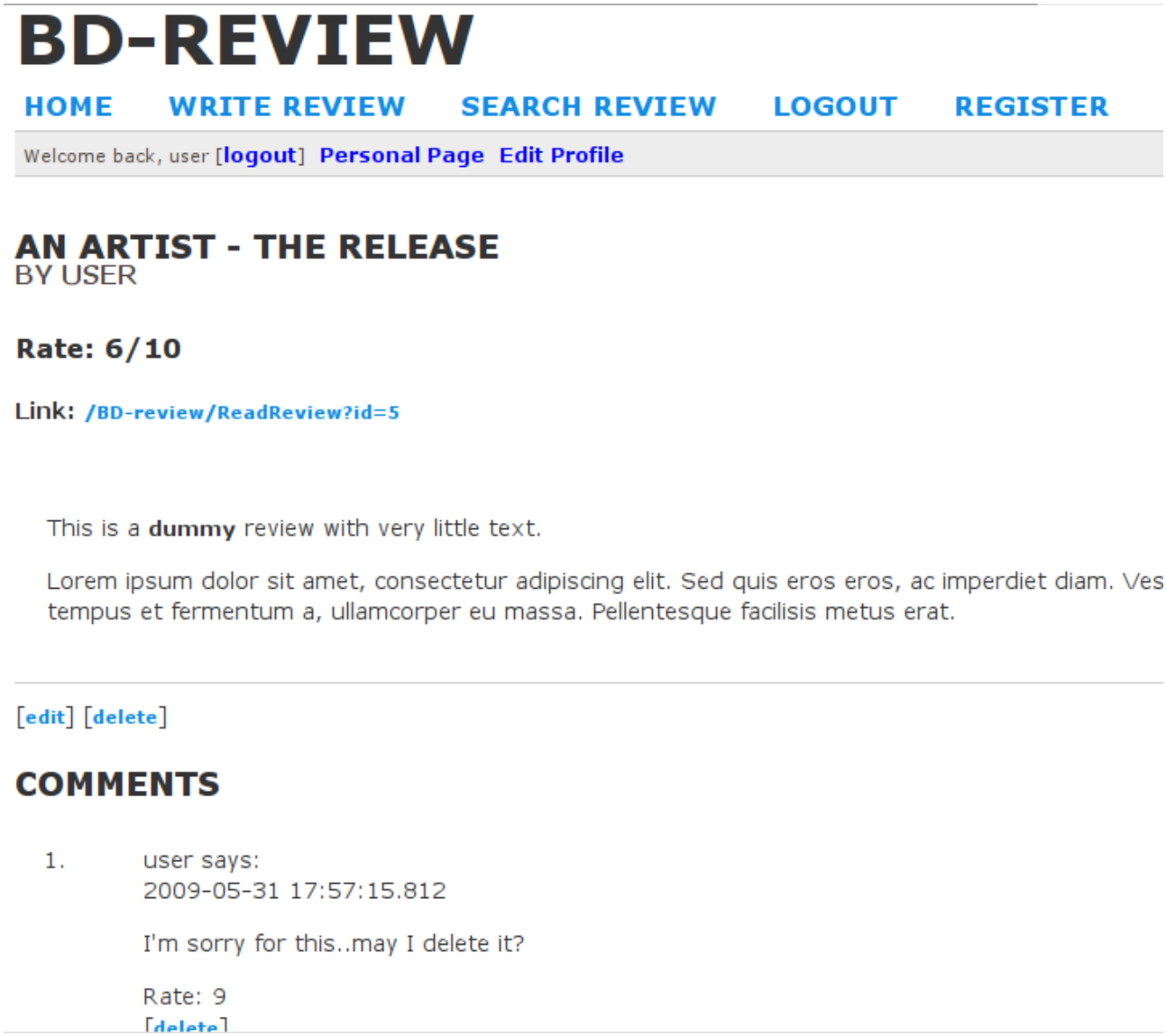
A new menu appears just after the main menu. This one is highlighted and contains links to protected parts of the website. The "LOGIN" link of the main menu is also replaced by a "LOGOUT" link.

The personal page of a user contains the most important information he needs: his last access, the new reviews made after his last access to the system and the reviews he wrote, with the possibility to either delete or modify them (or both). This depends on his rights.

Review Visualization

Reviews and comments are also very clean and clear. There is a tiny heading section where some information are given, regarding the Artist name and release, the author of the review, the rate given and the link to the review.

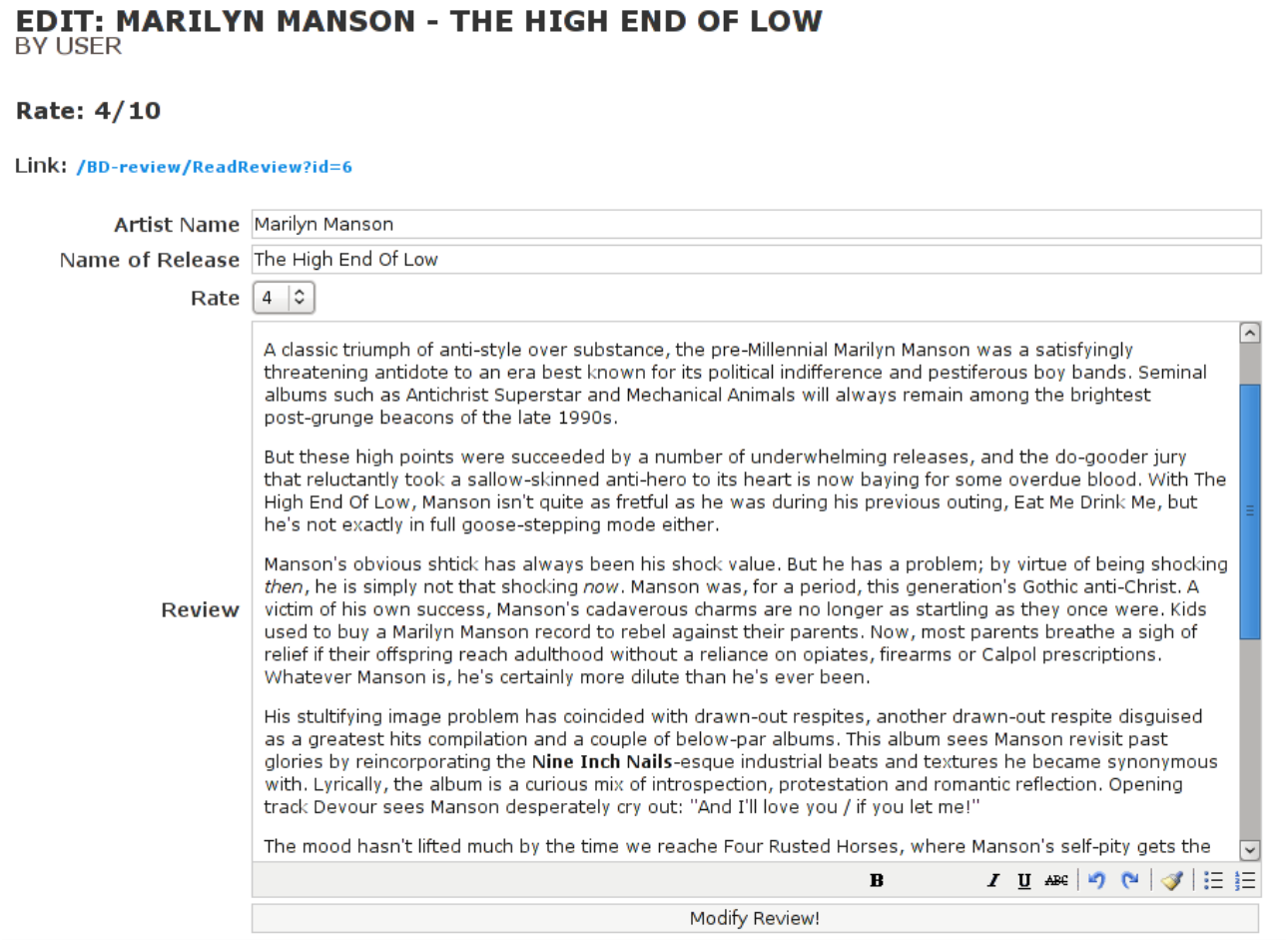
If the user has the right permissions, after each review and each comment may appear links to either edit or delete the write (or both of them).



Picture 9: A Review with a Comment

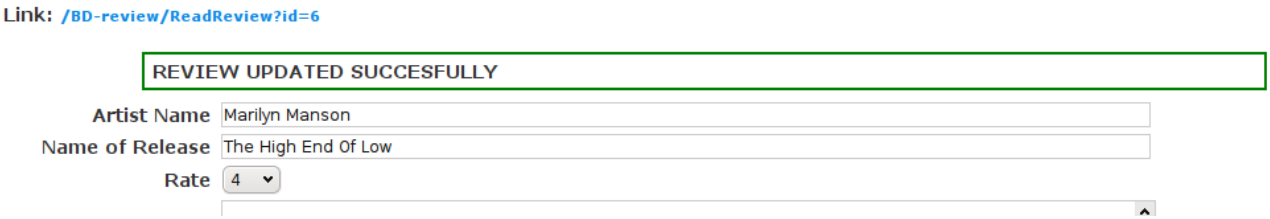
User input and Message System

In this page I will show a typical user input page and the message system of BD-review. In the picture we can see the modification of an existing review using the HTML editor TinyMCE



Picture 10: Modifying a Review

If after the submit everything worked fine, a message is printed before the form. The whole system uses a uniform way to display messages and error messages to the user.



Picture 11: Message System in Action

Problems found

- JSP/Servlets is a complex architecture compared to other web development frameworks. It feels like Java was not thought for web development. Quite everything must be created from scratch. On the other hand, more control is given to the developer that operates in a lower level. I also appreciated Filters and Listeners.
- The current Database connection solution is not a good choice. I simply use a class that returns the connection statically, creating it if needed. This is very bad in multi-threading applications like Servlets. It causes some rare NullPointerException exceptions sometimes. I read about connection pooling but I did not use them since this is a non real-life application.
- Filtering user input is very difficult, while is trivial with other web frameworks that provide elegant and powerful solutions.
- Object-Relational Mapping is missing and Hibernate is to difficult to be learned in a small amount of time.
- If this was a real-life application, there are a lot of features missing, such as a public profile for each user, a better layout, user activation system via e-mail, the possibility to upload the Cover Art image of a release, the possibility to ban a user without deleting him, encrypted passwords, powerful nickname and email validation systems etc.